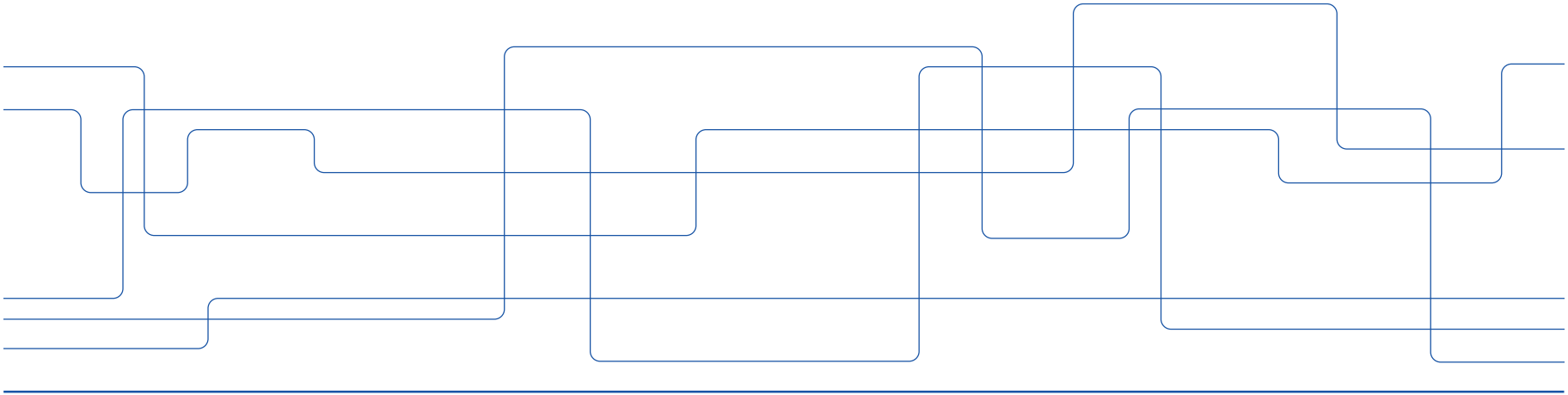
Generative Adversarial Network

Tianze Wang

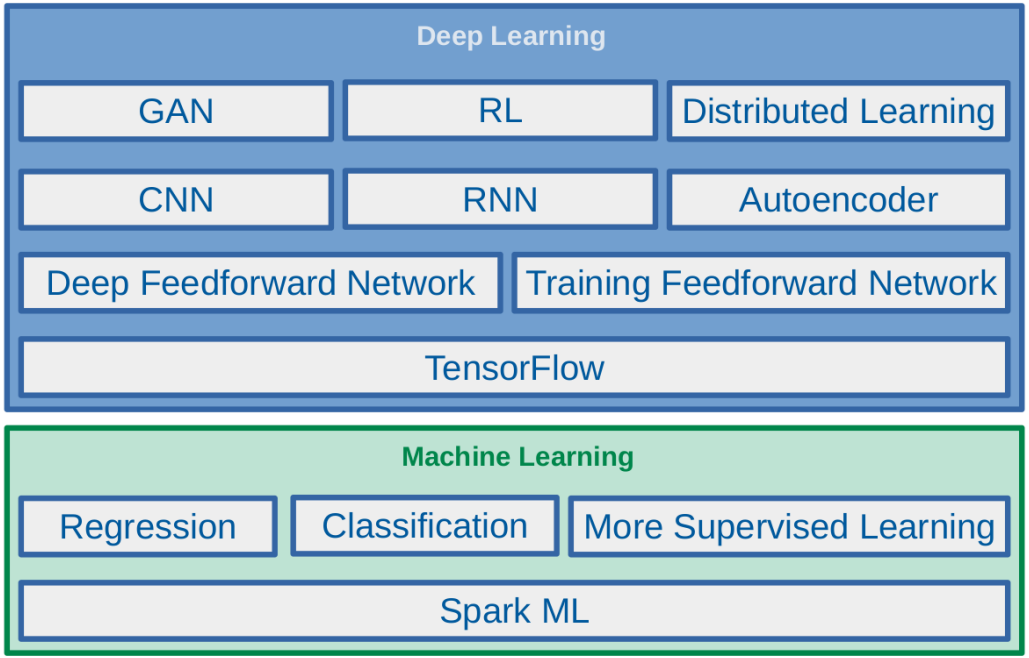
tianzew@kth.se



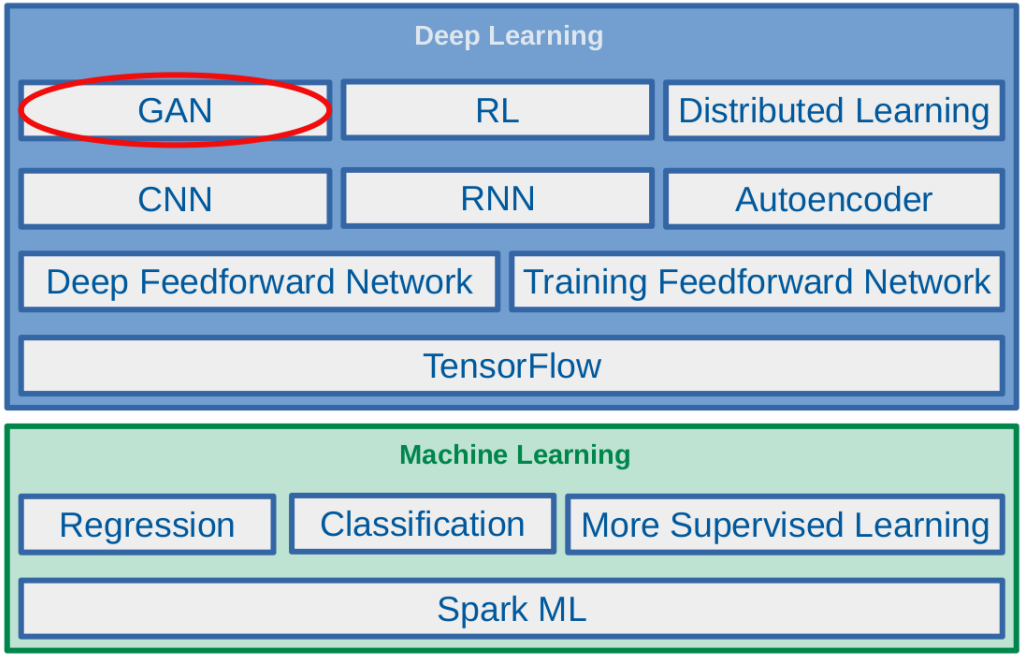
The Course Web Page

https://id2223kth.github.io

Where Are We?


Deep Learning
GAN
RL
Distributed Learning
CNN
RNN
Autoencoder
Deep Feedforward Network
Training Feedforward Network
TensorFlow
Machine Learning
Regression
Classification
More Supervised Learning
Spark ML

Where Are We?


Deep Learning
GAN
RL
Distributed Learning
CNN
RNN
Autoencoder
Deep Feedforward Network
Training Feedforward Network
TensorFlow
Machine Learning
Regression
Classification
More Supervised Learning
Spark ML

Let’s Start With What GANs Can Do

What GANs can do?

- Generating faces
- Generating Airbnb bedrooms
- Super resolution
- Colorization
- Turning a simple sketch into a photorealistic image
- Predicting the next frames in a video
- Augmenting a dataset
- and more…

An image generated by a StyleGAN that looks deceptively like a portrait of a young woman.

Quick overview of GANs

- **Generative Adversarial Networks (GANs)** are composed of two neural networks:
  - A ***generator***: tries to generate data that looks similar to the training data,
  - A ***discriminator*** that tries to tell real data from fake data.
- The ***generator*** and the ***discriminator*** compete against each other during training.
- ***Adversarial training*** is widely considered as one of the most important ideas in recent years.
- "*The most interesting idea in the last 10 years in Machine Learning.*"

by Yann LeCun in 2016

Generative Adversarial Network

GANs

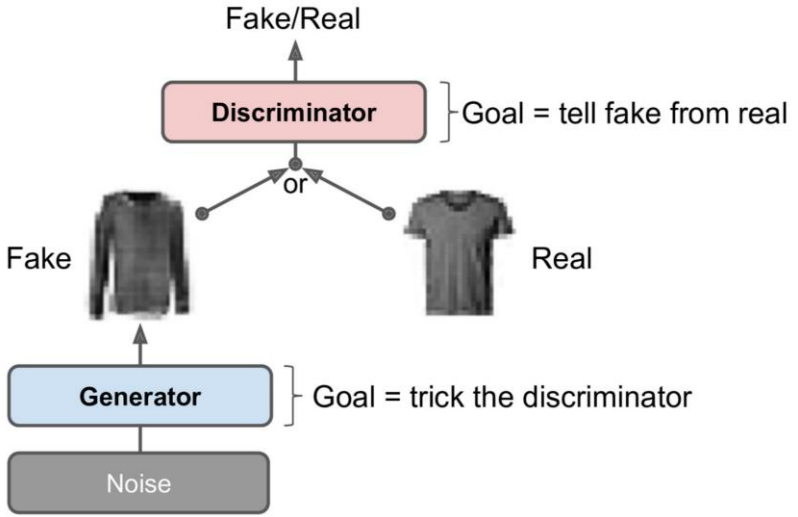
- GANs were proposed in 2014 by Ian Goodfellow et al.
- The idea behind GANs got researchers excited almost instantly.
- It took a few years to overcome some of the difficulties of training GANs.

The idea behind GANs

Make neural networks **compete against** each other in the hope that this competition will **push them to excel**.

Overall architecture of GANs

- A GAN is composed of two *neural networks*:
  - **Generator**:
    - > *Input: a random distribution (e.g., Gaussian)*
    - > *Output: some data (typically, an image)*
  - **Discriminator**:
    - > *Input: either a fake image from the generator or a real image from the training set*
    - > *Output: a guess on whether the input image is fake or real.*



A generative adversarial network

Training of GANs

- During training, the **generator** and the **discriminator** have opposite goals:
  - The discriminator tries to **tell** fake images from real images,
  - The generator tries to produce images that look real enough to **trick** the discriminator.
- Each training iteration is divided into two phases.

Training of GANs

In the first phase:

- Train the discriminator:
  - A batch of equal number of real images (sampled from the dataset) and fake images (produced by the generator) is passed to the discriminator.
  - The labels of the batch are set to 0 for fake images and 1 for real images.
  - Training is based on binary cross-entropy loss.
  - Backpropagation only optimizes the weights of the discriminator.

In the second phase:

- Train the generator:
  - First use the current generator to produce another batch containing only fake images.
  - The labels of the batch are set to 1. (we want the generator to produce images that the discriminator will wrongly believe to be real)
  - The weights of the discriminator are frozen during this step, so backpropagation only affects the weights of the generator.

A simple GAN for Fashion MNIST

```
codings_size = 30
# the generator
generator = keras.models.Sequential([
    # Scaled Exponential Linear Units (or SELUs)
    keras.layers.Dense(100, activation="selu", input_shape=[codings_size]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
    ])
# the discriminator
discriminator = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(1, activation="sigmoid")
])
# the GAN model
gan = keras.models.Sequential([generator, discriminator])
# compile the model
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

A simple GAN for Fashion MNIST

```
# the Dataset
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
# our special training function
def train_gan(gan, dataset, batch_size, codings_size, n_epochs=50):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
            # phase 1 - training the discriminator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator(noise)
            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
            discriminator.trainable = True
            discriminator.train_on_batch(X_fake_and_real, y1)
            # phase 2 - training the generator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)
            discriminator.trainable = False
            gan.train_on_batch(noise, y2)
# train the GAN model
train_gan(gan, dataset, batch_size, codings_size)
```

Images generated by the GAN

Images generated by the GAN after one epoch of training

What next?

- Build a GAN model
- Train for many epochs
- ?????
- Good RESULTS!

Difficulties of Training GANs

Difficulties of Training GANs

- During training, the generator and the discriminator constantly try to outsmart each other.
- As training goes on, the networks may end up in a state that game theorists call a **Nash equilibrium**.

Nash Equilibrium

- In game theory, the **Nash equilibrium**, named after the mathematician John Forbes Nash Jr., is a proposed solution of a non-cooperative game involving two or more players in which each player is assumed to know the equilibrium strategies of the other players, and no player has anything to gain by changing only their own strategy.
- For example, a Nash equilibrium is reached when everyone drives on the left side of the road: no driver would be better off being the only one to switch sides.
- Different initial states and dynamics may lead to one equilibrium or the other.

How does this apply to GANs

- It has been demonstrated that a GAN can only reach a single Nash equilibrium.
- In that case, the generator produces perfectly realistic images, and the discriminator is forced to guess (50% real, 50% fake).
- Unfortunately, nothing guarantees that the equilibrium will ever be reached.
- The biggest difficulty is called **mode collapse**:
  - when the generator's outputs gradually become less diverse.

Mode Collapse

- The generator gets better at producing convincing shoes than any other class.
- This will encourage it to produce even more images of shoes. Gradually, it will forget how to produce anything else.
- Meanwhile, the only fake images that the discriminator will see will be shoes, so it will also forget how to discriminate fake images of other classes.
- Eventually, when the discriminator manages to discriminate the fake shoes from the real ones, the generator will be forced to move to another class.
- The GAN may gradually cycle across a few classes, never really becoming very good at any of them.

Training might be problematic as well

- Because the generator and the discriminator are constantly pushing against each other, their parameters may end up oscillating and becoming unstable.
- Training may begin properly, then suddenly diverge for no apparent reason, due to these instabilities.
- GANs are very sensitive to the hyperparameters since many factors can contribute to the complex dynamics.

How to Deal with the Difficulties?

Experience Replay

- A common technique to train GANs:
  - Store the images produced by the generator at each iteration in **a replay buffer** (gradually dropping older generated images).
  - Train the discriminator using real images plus fake images drawn **from this buffer** (rather than only using fake images produced by the current generator).

- **Experience replay** reduces the chances that the discriminator will overfit the latest generator's output.

Mini-batch Discrimination

- Another common technique that:
  - Measures how similar images are across the batch and provide this statistics to the discriminator.
  - so that the discriminator can easily reject a batch of images that lack diversity.

- **Mini-batch discrimination** encourages the generator to produce a greater variety of images, thus reducing the chance of model collapse.

Deep Convolutional GANs

Deep Convolutional GANs (DCGANs)

- The original GAN paper in 2014 experimented with convolutional layers, but only tried to generate small images.
- Build GANs based on deeper convolutional nets for larger images is tricky, as training was very unstable.
- But in late 2015 Alec Radford et al. proposed *deep convolutional GANs* (**DCGANs**) after experimenting with many different architectures and hyperparameters.

Radford, A.; Metz, L. & Chintala, S. (2015), 'Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks' , cite arxiv:1511.06434Comment: Under review as a conference paper at ICLR 2016 .

Deep Convolutional GANs (DCGANs)

The main guidelines they proposed for building stable convolutional GANs:

- Replace any pooling layers with strided convolutions (in the discriminator) and transposed convolutions (in the generator).
- Use Batch Normalization in both the generator and the discriminator, except in the generator's output layer and the discriminator's input layer.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in the generator for all layers except the output layer, which should use tanh.
- Use leaky ReLU activation in the discriminator for all layers.

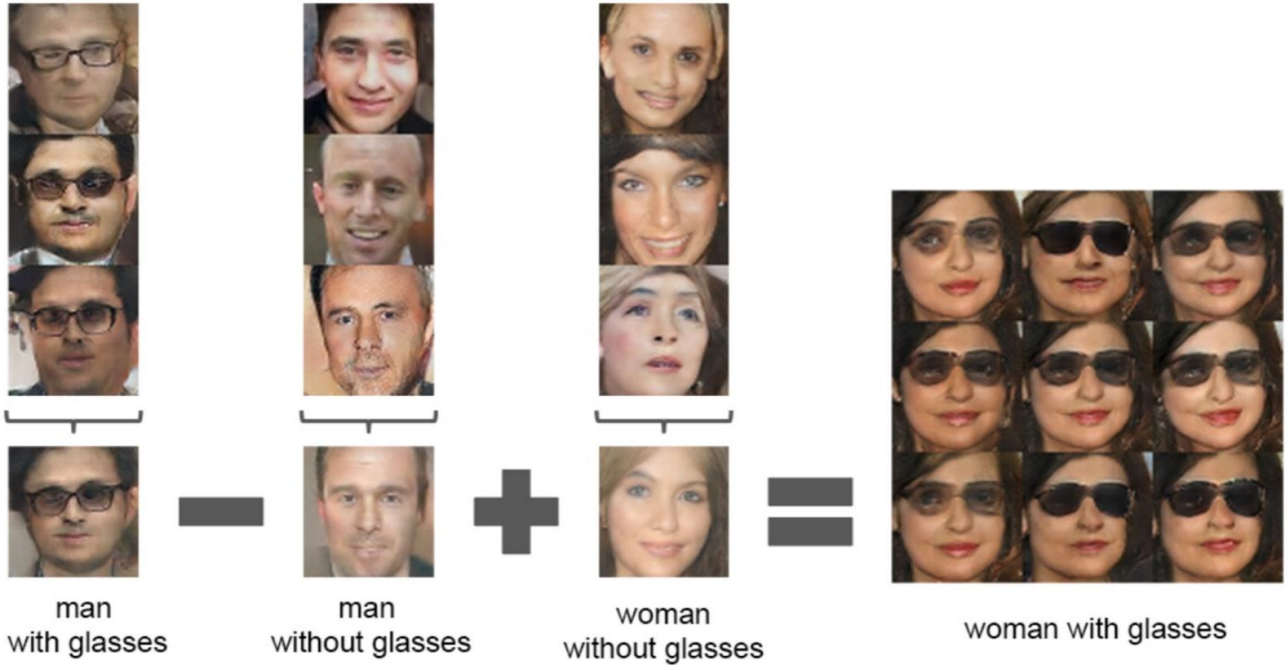
DCGAN for Fashion MNIST

```
codings_size = 100
# the generator
generator = keras.models.Sequential([
    keras.layers.Dense(7 * 7 * 128, input_shape=[codings_size]),
    keras.layers.Reshape([7, 7, 128]),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding="same", activation="selu"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2, padding="same", activation="tanh")
])
# the discriminator
discriminator = keras.models.Sequential([
    keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
    activation=keras.layers.LeakyReLU(0.2),
    input_shape=[28, 28, 1]),
    keras.layers.Dropout(0.4),
    keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
    activation=keras.layers.LeakyReLU(0.2)),
    keras.layers.Dropout(0.4),
    keras.layers.Flatten(),
    keras.layers.Dense(1, activation="sigmoid")
])
# the gan model
gan = keras.models.Sequential([generator, discriminator])
# need to reshape the data
X_train = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # reshape and rescale
```

DCGAN for Fashion MNIST

Images generated by the DCGAN after 50 epochs of training

DCGAN for Fashion MNIST



Vector arithmetic for visual concepts (part of figure 7 from the DCGAN paper)

Limitations of DCGANs

- DCGANs aren’t perfect, though.
- For example, when you try to generate very large images using DCGANs, you often end up with locally convincing features but overall inconsistencies (such as shirts with one sleeve much longer than the other).

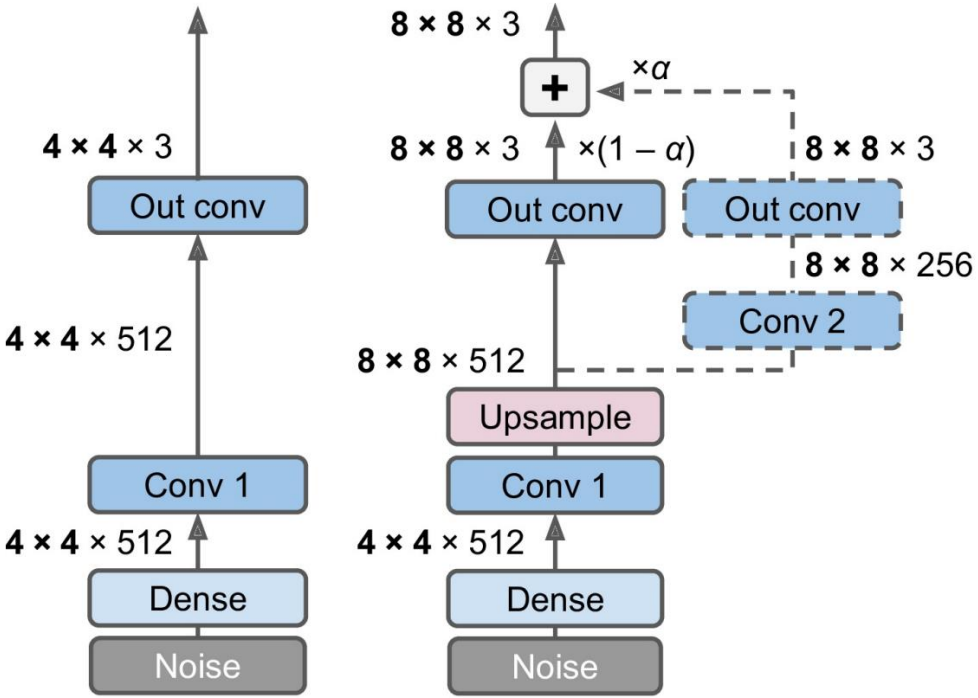
Progressive Growing of GANs

An important technique

- Tero Karras et al. suggested generating small images at the beginning of training, then gradually adding convolutional layers to both the generator and the discriminator to produce larger and larger images (4 × 4, 8 × 8, 16 × 16, …, 512 × 512, 1,024 × 1,024).
- This approach resembles greedy layer-wise training of stacked autoencoders.
- The extra layers get added at the end of the generator and at the beginning of the discriminator, and previously trained layers remain trainable.

Tero Karras et al., “Progressive Growing of GANs for Improved Quality, Stability, and Variation,” *Proceedings of the International Conference on Learning Representations* (2018)

Progressive Growing of GAN



Progressive growing GAN: a GAN generator outputs 4 × 4 color images (left); we extend it to output 8 × 8 images (right)

Minibatch Standard Deviation Layer

- Added near the end of the discriminator. For each position in the inputs, it computes the standard deviation across all channels and all instances in the batch.
- These standard deviations are then averaged across all points to get a single value.
- Finally, an extra feature map is added to each instance in the batch and filled with the computed value.
- How does this help? Well, if the generator produces images with little variety, then there will be a small standard deviation across feature maps in the discriminator. Thanks to this layer, the discriminator will have easy access to this statistic, making it less likely to be fooled by a generator that produces too little diversity. This will encourage the generator to produce more diverse outputs, reducing the risk of mode collapse.

Equalized Learning Rate

- Initializes all weights using a simple Gaussian distribution with mean 0 and standard deviation 1 rather than using He initialization.
- However, the weights are scaled down at runtime (i.e., every time the layer is executed) by the same factor as in He initialization: they are divided by $\sqrt{\frac{2}{n_{inputs}}}$, where n_{inputs} is the number of inputs to the layer.
- The paper demonstrated that this technique significantly improved the GAN's performance when using RMSProp, Adam, or other adaptive gradient optimizers.
- By rescaling the weights as part of the model itself rather than just rescaling them upon initialization, this approach ensures that the dynamic range is the same for all parameters, throughout training, so they all learn at the same speed. This both speeds up and stabilizes training.

Pixelwise Normalization Layer

- Added after each convolutional layer in the generator. It normalizes each activation based on all the activations in the same image and at the same location, but across all channels (dividing by the square root of the mean squared activation).
- This technique avoids explosions in the activations due to excessive competition between the generator and the discriminator.

Amazing Results

- The combination of all these techniques allowed the authors to generate extremely good results (https://www.youtube.com/watch?v=G06dEcZ-QTg) .
- Evaluation is one of the big challenges when working with GANs:
  - Auto-evaluation is tricky as evaluation is subjective
  - Using human raters is costly and time-consuming
  - The authors proposed to measure the similarity between the local image structure of the generate image and the training images.

StyleGANs

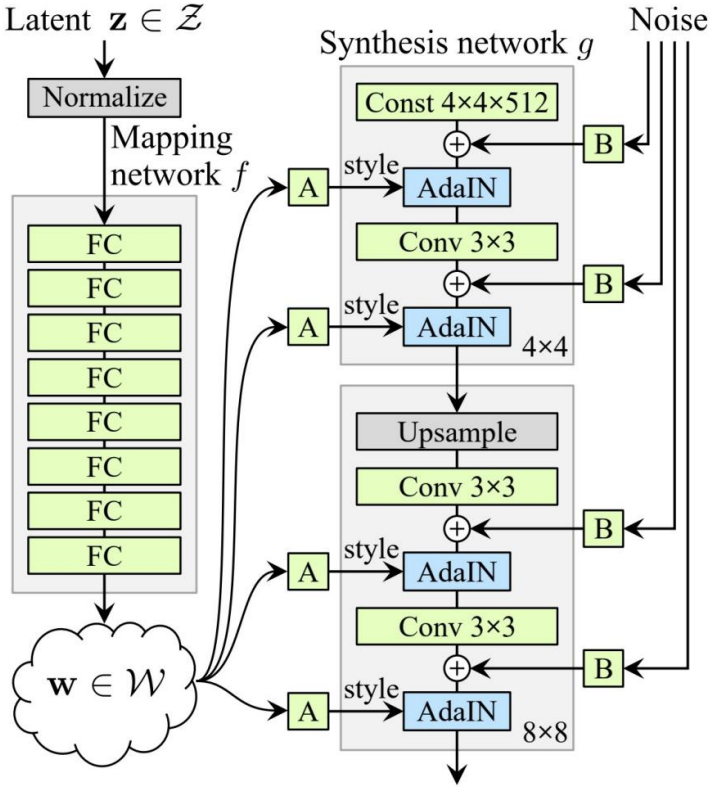
StyleGANs

- The authors used *style transfer* techniques in the generator to ensure that the generated images have the same local structure as the training images, at every scale, greatly improving the quality of the generated images.
- StyleGANs is composed of two networks:
  - Mapping Network
  - Synthesis Network
- The discriminator and the loss function were not modified, only the generator.

StyleGANs: Mapping Network

Mapping Network:

- An eight-layer MLP that maps the latent representations **z** (i.e., the codings) to a vector **w**.
- This vector is then sent through multiple *affine transformations* which produces multiple vectors.
- These vectors control the style of the generated image at different levels, from fine-grained texture (e.g., hair color) to high-level features (e.g., adult or child). In short, the mapping network maps the codings to multiple style vectors.

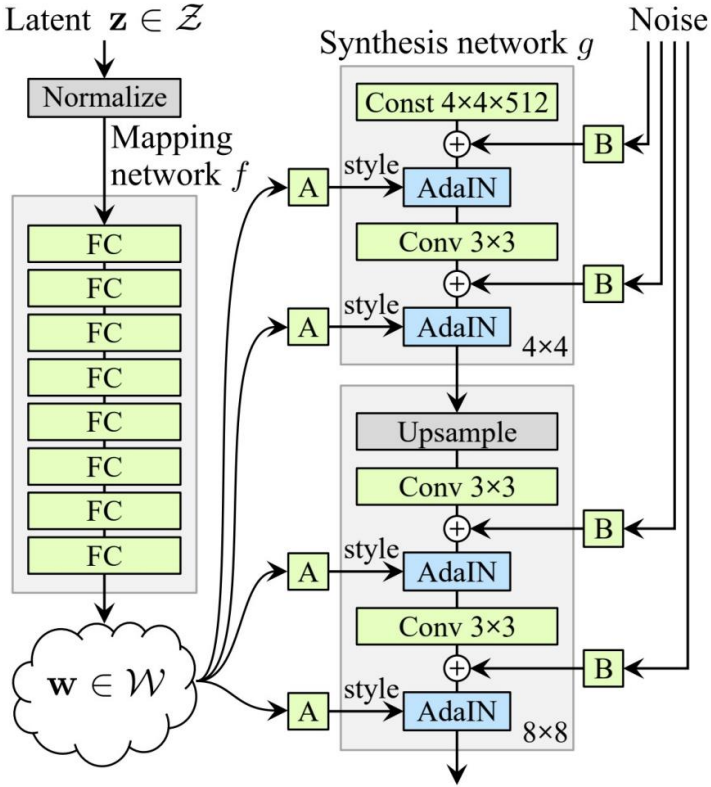


StyleGAN's generator architecture (part of figure 1 from the StyleGAN paper)

StyleGANs: Synthesis Network

Synthesis Network

- Responsible for generating the images.
- It has a constant learned input.
- It processes this input through multiple convolutional and upsampling layers, but there are two twists:
  - some noise is added to the input and to all the outputs of the convolutional layers
  - each noise layer is followed by an *Adaptive Instance Normalization* (AdaIN) layer: it standardizes each feature map independently, then it uses the style vector to determine the scale and offset of each feature map.



StyleGAN's generator architecture (part of figure 1 from the StyleGAN paper)

Summary

Summary

- What are GANs?
- Main difficulties with adversarial training
- Main techniques to work around these difficulties
  - Experience replay
  - Mini-batch discrimination
- Deep convolutional GANs
- Progressive Growing of GANs
- StyleGANs

Questions?